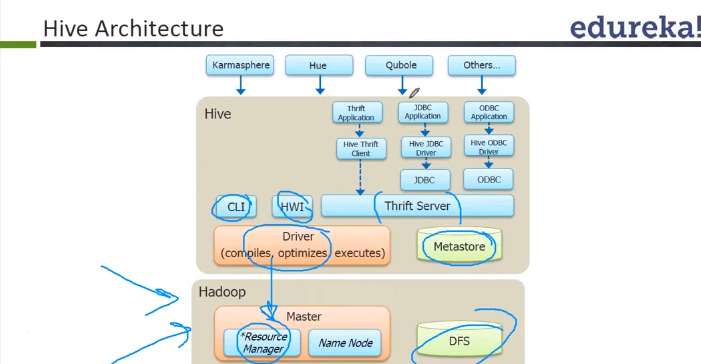
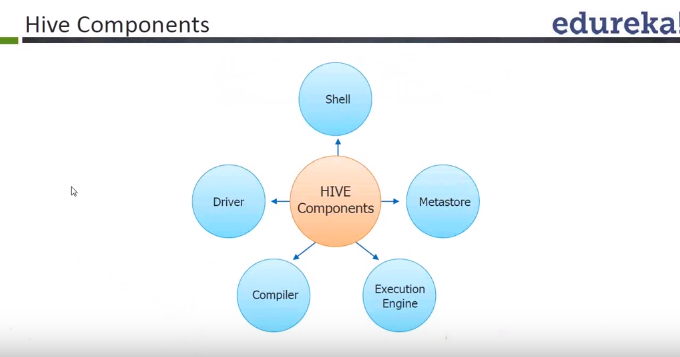
1. **Hive Architecture**

****

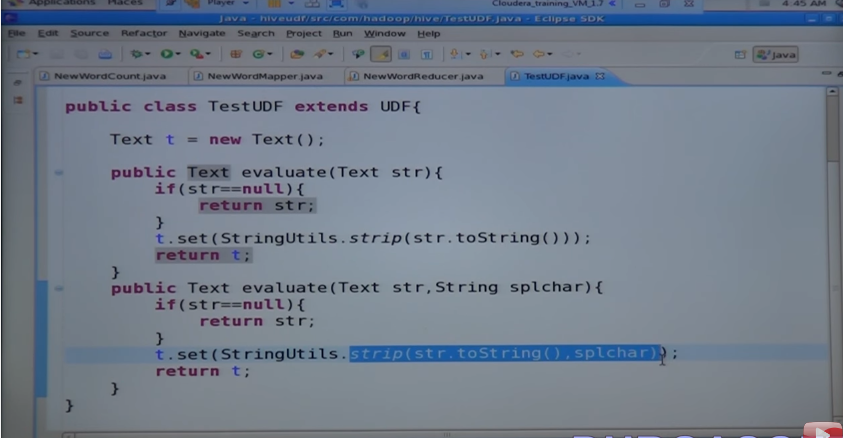
****

**Thrift Server – Allows to connect to any database using JDBC or ODBC.**

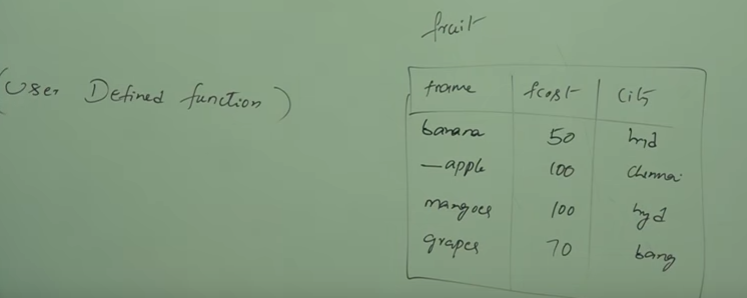
**Metastore – Hive Metadata (Schemas) are stored on Metastore. Metastore itself is a database.**

1. **User Defined Functions in Hive**

**Sample UDF to strip leading and trailing spaces or any leading or trailing characters**

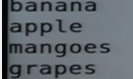
****

**t**

****

**Rules and Steps:**

1. Our UDF class should extend UDF Abstract class
2. Our UDF class must have atleast one evaluate method
3. Compile the above java file
4. Create the Jar file for keeping the class file
5. Add the jar file to hive class path
6. Create temporary function



1. **Regular Expression Function**

**regexp\_extract**

regexp\_extract('foothebar', 'foo(.\*?)(bar)', 2) returns bar.

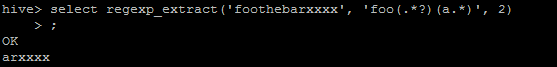
The index parameter is the capture group, which is an integer that can take the following values:

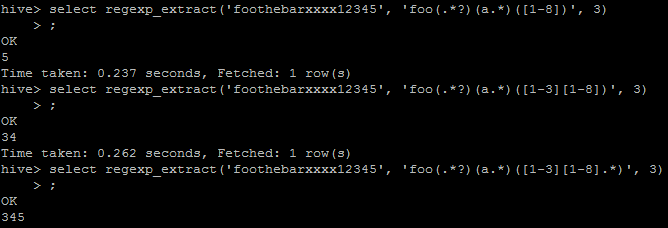
* 0: the entire match, in my example it would be foothebar
* 1: the first group, in my example it would be the
* 2: the second group, in my example it would be bar
* n: the nth group. If n is bigger than the actual number of groups defined in your regexp, your Hive query will fail.

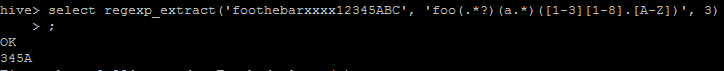
In your example, regexp\_extract(input, '[0-9]\*', 0), your are looking for the whole match for your column identified by input and starting with a numerical value.

Here are a few examples:

* regexp\_extract('9eleven', '[0-9]\*', 0) -> returns 9
* regexp\_extract('9eleven', '[0-9]\*', 1) -> query fails
* regexp\_extract('911test', '[0-9]\*', 0) -> returns 911
* regexp\_extract('911test', '[0-9]\*', 1) -> query fails
* regexp\_extract('eleven', '[0-9]\*', 0) -> returns empty string
* regexp\_extract('test911', '[0-9]\*', 0) -> returns empty string

****

****

****

regexp\_replace : Find and Replace

I am using hive 0.13! I want to find multiple tokens like "hip hop" and "rock music" in my data and replace them with "hiphop" and "rockmusic" - basically replace them without white space. I have used the regexp\_replace function in hive. Below is my query and it works great for above 2 examples.

drop table vp\_hiphop;

create table vp\_hiphop as

select userid, ntext,

regexp\_replace(regexp\_replace(ntext, 'hip hop', 'hiphop'), 'rock music', 'rockmusic') as ntext1

from vp\_nlp\_protext\_males

;

select userid, ntext,

regexp\_replace(ntext, '(hip hop)|(rock music)', '(hiphop)|(rockmusic)' ) as ntext1

1. **CAST Function**

This function is used to convert from one data type to another. The only type conversion function is CAST.

The syntax of CAST is **CAST( expr as <type> )**

**Example : CAST(regexp\_replace(erdat, '\\.', '')as Int)**

**Note** : In the above example, erdat is date column, regexp\_replace will replace all dots ‘.’ From the date with nothing, I mean it will remove the dots “.” and convert its type to integer using CAST to compare against another integer column.

**Complex Query using Regular Expression replace , CAST and IF condition :**

SELECT \* FROM d\_core\_data.billing

WHERE CAST(regexp\_replace(erdat, '\\.', '') as int) >= CAST(CONCAT(CAST(IF(month(from\_unixtime(unix\_timestamp())) < 10, year(from\_unixtime(unix\_timestamp()))-2, year(from\_unixtime(unix\_timestamp()))-1) as String),"1001") as int)

Note : selecting all columns from table billing where

Using regexp\_replace, it will remove all the dots from the date column erdat and compare it against formed date

How is date formed “ extracts the current month using month function and using IF condition checks if month is less than 10, if yes, subtracts 2 from current year else subtracts 1 from current year and concats the year with string “1000” and converts the type to integer for comparision

1. **Storage File formats in Hive :**

Like SQL, [HiveQL](https://acadgild.com/big-data/big-data-development-training-certification) handles structured data only. By default, Hive has derby database to store the data in it. We can configure Hive with [MySQL](https://acadgild.com/big-data/big-data-development-training-certification) database. As mentioned HiveQL can handle only structured data. Data is eventually stored in files. There are some specific file formats which Hive can handle such as:  
• TEXTFILE  
• SEQUENCEFILE  
• RCFILE  
• ORCFILE

**SEQUENCEFILE -** Sequence files are flat files consisting of binary key-value pairs. When Hive converts queries to MapReduce jobs, it decides on the appropriate key-value pairs to be used for a given record. Sequence files are in the binary format which are able to split and the main use of these files is to club two or more smaller files and make them as a one sequence file.

In Hive we can create a sequence file by specifying STORED AS SEQUENCEFILE in the end of a CREATE TABLE statement.

There are three types of sequence files:

* Uncompressed key/value records.
* Record compressed key/value records – only ‘values’ are compressed here
* Block compressed key/value records – both keys and values are collected in ‘blocks’ separately and compressed. The size of the ‘block’ is configurable.

Hive has its own SEQUENCEFILE reader and SEQUENCEFILE writer for reading and writing through sequence files.

create table olympic\_sequencefile(athelete STRING,age INT,country STRING,year STRING,closing STRING,sport STRING,gold INT,silver INT,bronze INT,total INT) row format delimited fields terminated by '\t' stored as sequencefile

INSERT OVERWRITE TABLE olympic\_sequencefile

SELECT \* FROM olympic

**RC File**

RCFILE stands of Record Columnar File which is another type of binary file format which offers high compression rate on the top of the rows.  
RCFILE is used when we want to perform operations on multiple rows at a time.  
RCFILEs are flat files consisting of binary key/value pairs, which shares much similarity with SEQUENCEFILE. RCFILE stores columns of a table in form of record in a columnar manner. It first partitions rows horizontally into row splits and then it vertically partitions each row split in a columnar way. RCFILE first stores the metadata of a row split, as the key part of a record, and all the data of a row split as the value part. This means that RCFILE encourages column oriented storage rather than row oriented storage.  
This column oriented storage is very useful while performing analytics. It is easy to perform analytics when we “hive’ a column oriented storage type.  
Facebook uses RCFILE as its default file format for storing of data in their data warehouse as they perform different types of analytics using Hive.

create table olympic\_rcfile(athelete STRING,age INT,country STRING,year STRING,closing STRING,sport STRING,gold INT,silver INT,bronze INT,total INT) row format delimited fields terminated by '\t' stored as rcfile

INSERT OVERWRITE TABLE olympic\_rcfile

SELECT \* FROM olympic;

**ORCFILE**  
ORC stands for Optimized Row Columnar which means it can store data in an optimized way than the other file formats. ORC reduces the size of the original data up to 75%. As a result the speed of data processing also increases. ORC shows better performance than Text, Sequence and RC file formats.  
An ORC file contains rows data in groups called as Stripes along with a file footer. ORC format improves the performance when Hive is processing the data.

create table olympic\_orcfile(athelete STRING,age INT,country STRING,year STRING,closing STRING,sport STRING,gold INT,silver INT,bronze INT,total INT) row format delimited fields terminated by '\t' stored as orcfile;

INSERT OVERWRITE TABLE olympic\_orcfile

SELECT \* FROM olympic;

1. **SerDe Overview**

SerDe is short for Serializer/Deserializer. Hive uses the SerDe interface for IO. The interface handles both serialization and deserialization and also interpreting the results of serialization as individual fields for processing.

A SerDe allows Hive to read in data from a table, and write it back out to HDFS in any custom format. Anyone can write their own SerDe for their own data formats.

See [Hive SerDe](https://cwiki.apache.org/confluence/display/Hive/DeveloperGuide#DeveloperGuide-HiveSerDe) for an introduction to SerDes.

when querying a table, a SerDe will **deserialize a row of data from the bytes in the file to objects used internally by Hive to operate on that row of data**. when performing an INSERT or CTAS (see “Importing Data” on page 441), the table’s SerDe will **serialize Hive’s internal representation of a row of data into the bytes that are written to the output file**.

* 1. **Processing Json File**

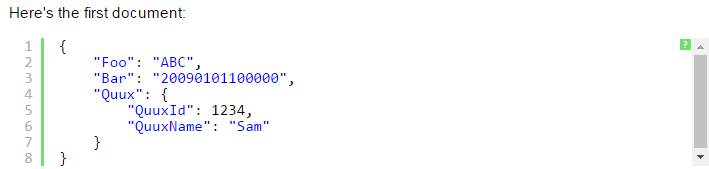
I've only been playing with Hive about two weeks now, but here's what I found with respect to using complex JSON documents with Hive.

Hive has two built-in functions, get\_json\_object and json\_tuple, for dealing with JSON. There are also a couple of JSON SerDe's (Serializer/Deserializers) for Hive. I like this one the best:<https://github.com/rcongiu/Hive-JSON-Serde>

I will document using these three options here.

Let's start with a simple JSON document and then move to a complex document with nested subdocuments and arrays of subdocuments.

Here's the first document:



We are going to store this as a Text document, so it is best to have the whole JSON entry on a single line in the text file you point the Hive table to.

Here it is on one line for easy copy and pasting:

|  |  |
| --- | --- |
|  | {"Foo":"ABC","Bar":"20090101100000","Quux":{"QuuxId":1234,"QuuxName":"Sam"}} |

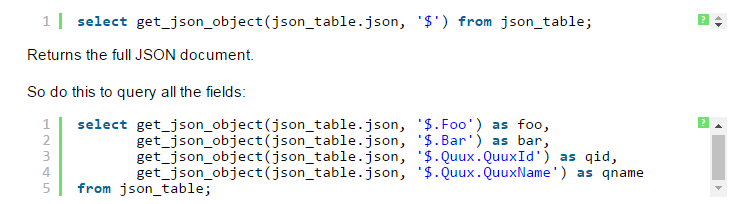
Let's create a Hive table to reference this. I've put the above document in a file called simple.json:

|  |  |
| --- | --- |
|  | CREATE TABLE json\_table ( json string );  LOAD DATA LOCAL INPATH  '/tmp/simple.json' INTO TABLE json\_table; |

Since there are no delimiters, we leave off the ROW FORMAT section of the table DDL

#### Built in function #1: get\_json\_object

The get\_json\_object takes two arguments: tablename.fieldname and the JSON field to parse, where '$' represents the root of the document.



You should get the output:

foo bar qid qname

ABC 20090101100000 1234 Sam

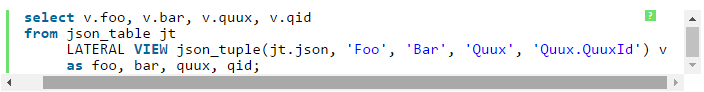
(Note: to get the header fields, enter set hive.cli.print.header=true at the hive prompt or in your$HOME/.hiverc file.)

This works and has a nice JavaScript like "dotted" notation, but notice that you have to parse the same document once for every field you want to pull out of your JSON document, so it is rather inefficient.

The Hive wiki recommends using json\_tuple for this reason.

#### Built in function #2: json\_tuple

So let's see what json\_tuple looks like. It has the benefit of being able to pass in multiple fields, but it only works to a single level deep. You also need to use Hive's slightly odd LATERAL VIEW notation:



This returns:

foo bar quux qid

ABC 20090101100000 {"QuuxId":1234,"QuuxName":"Sam"} NULL

It doesn't know how to look inside the Quux subdocument. And this is where json\_tuple gets clunky fast - you have to create another lateral view for each subdocument you want to descend into:

[?](http://thornydev.blogspot.in/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
|  | select v1.foo, v1.bar, v2.qid, v2.qname  from json\_table jt       LATERAL VIEW json\_tuple(jt.json, 'Foo', 'Bar', 'Quux') v1       as foo, bar, quux       LATERAL VIEW json\_tuple(v1.quux, 'QuuxId', 'QuuxName') v2       as qid, qname; |

This gives us the output we want:

foo bar qid qname

ABC 20090101100000 1234 Sam

With a complicated highly nested JSON doc, json\_tuple is also quite inefficient and clunky as hell. So let's turn to a custom SerDe to solve this problem.

#### The best option: rcongiu's Hive-JSON SerDe

A SerDe is a better choice than a json function (UDF) for at least two reasons:

1. it only has to parse each JSON record once
2. you can define the JSON schema in the Hive table schema, making it much easier to issue queries against.

I reviewed a couple of SerDe's and by far the best one I've found is [rcongiu's Hive-JSON SerDe](https://github.com/rcongiu/Hive-JSON-Serde).

To get that SerDe, clone the project from GitHub and run mvn package. It creates a json-serde-1.1.6.jar in the target directory. If you have a place you like to put your jars for runtime referencing move it there.

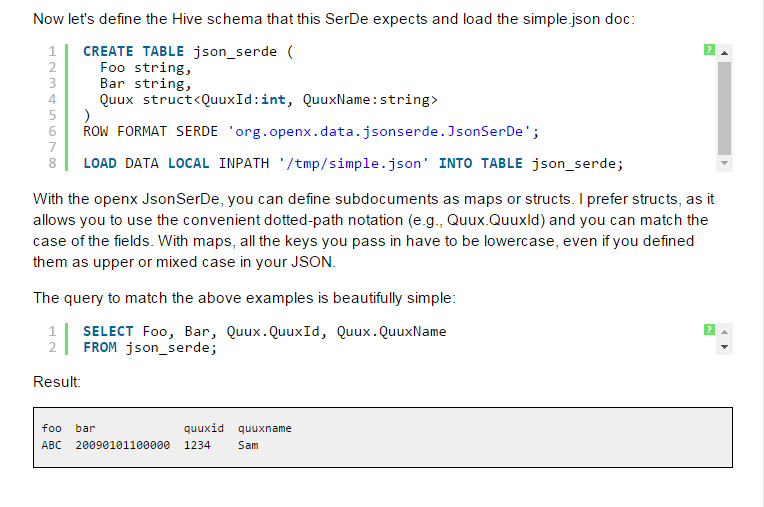
Then tell Hive about it with:

ADD JAR /path/to/json-serde-1.1.6.jar;

You can do this either at the hive prompt or put it in your $HOME/.hiverc file.

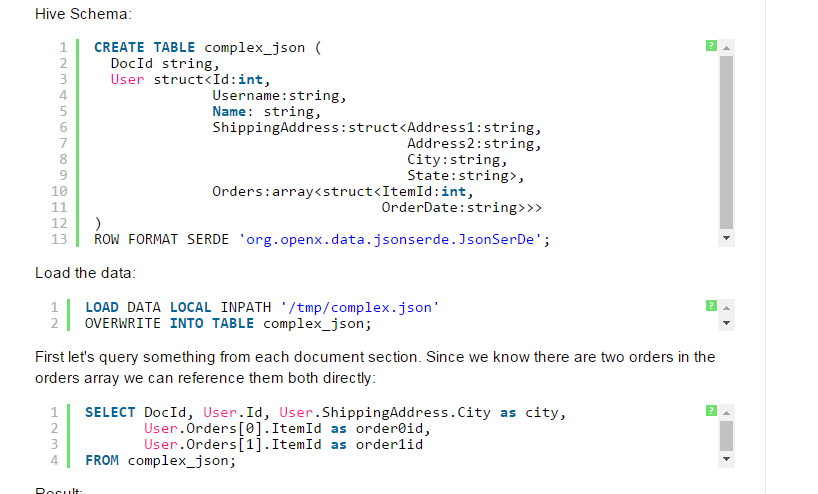
|  |  |
| --- | --- |
|  | {"Foo":"ABC","Bar":"20090101100000","Quux":{"QuuxId":1234,"QuuxName":"Sam"}} |

Now let's define the Hive schema that this SerDe expects and load the simple.json dochive> create table json\_serde (Foo String,Bar String, Quux struct <QuuxID:int, QuuxName:String>) ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'



**Eg : create table json\_serde (Foo String,Bar String, Quux struct <QuuxID:int, QuuxName:String>) ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'**





Result:

docid id city order0id order1id

ABC 1234 Durham 6789 4352

* 1. **Load XML file into Hive Table using xpath**

**Here is a sample input XML file:**  
  
**$cat employees.xml**  
<employee>  
<id>1</id>  
<name>Satish Kumar</name>  
<designation>Technical Lead</designation>  
</employee>  
<employee>  
<id>2</id>  
<name>Ramya</name>  
<designation>Testing</designation>  
</employee>  
 **Step:1 Bring each record to one line, by executing below command**

**$cat employees.xml | tr -d '&' | tr '\n' ' ' | tr '\r' ' ' | sed 's|</employee>|</employee>\n|g' | grep -v '^\s\*$' > employees\_records.xml**

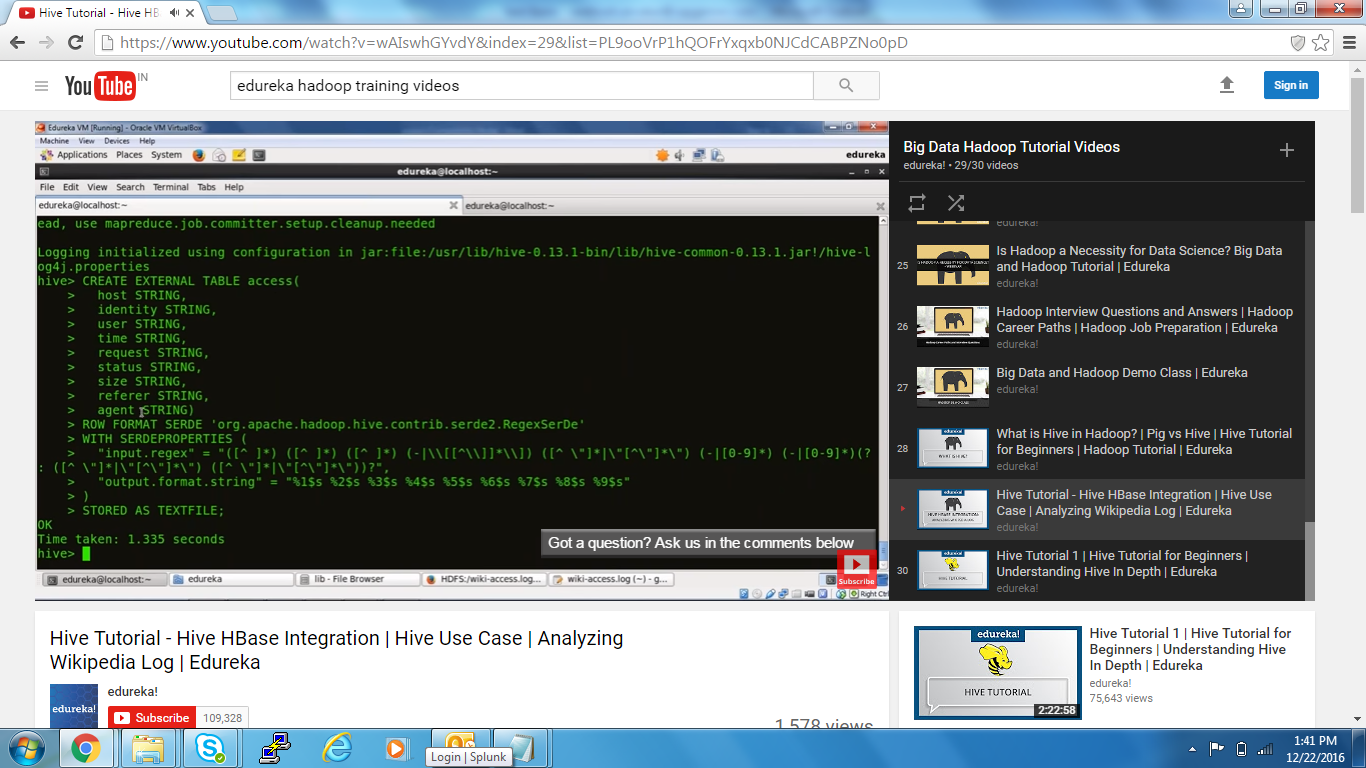
**$cat employees\_records.xml**

<employee> <id>1</id> <name>Satish Kumar</name> <designation>Technical Lead</designation> </employee>

<employee> <id>2</id> <name>Ramya</name> <designation>Testing</designation> </employee>

**Step:2 Load the file to HDFS**  
 **$hadoop fs -mkdir /user/hive/sample-xml-inputs**  
 **$hadoop fs -put employees\_records.xml /user/hive/sample-xml-inputs**  
 **$hadoop fs -cat /user/hive/sample-xml-inputs/employees\_records.xml**  
<employee> <id>1</id> <name>Satish Kumar</name><designation>Technical Lead</designation> </employee>  
<employee> <id>2</id> <name>Ramya</name> <designation>Testing</designation> </employee>  
  
**Step:3 Create a Hive table and point to xml file**  
 **hive>create external table xml\_table\_org( xmldata string) LOCATION '/user/hive/sample-xml-inputs/';**  
 **hive> select \* from xml\_table\_org;**  
**OK**  
<employee> <id>1</id> <name>Satish Kumar</name> <designation>Technical Lead</designation> </employee>  
<employee> <id>2</id> <name>Ramya</name> <designation>Testing</designation> </employee>  
  
Time taken: 0.179 seconds  
  
**Step 4: From the stage table we can query the elements and load it to other table.**  
 **hive> CREATE TABLE xml\_table AS SELECT xpath\_int(xmldata,'employee/id'),xpath\_string(xmldata,'employee/name'),xpath\_string(xmldata,'employee/designation') FROM xml\_table\_org;**  
Total MapReduce jobs = 3  
Launching Job 1 out of 3  
Number of reduce tasks is set to 0 since there's no reduce operator  
Starting Job = job\_201506301103\_0001, Tracking URL = http://0.0.0.0:50030/jobdetails.jsp?jobid=job\_201506301103\_0001  
Kill Command = /usr/lib/hadoop/bin/hadoop job  -kill job\_201506301103\_0001  
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0  
2015-06-30 11:23:10,969 Stage-1 map = 0%,  reduce = 0%  
2015-06-30 11:23:18,040 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 0.95 sec  
2015-06-30 11:23:19,058 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 0.95 sec  
2015-06-30 11:23:20,067 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 0.95 sec  
2015-06-30 11:23:21,079 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 0.95 sec  
MapReduce Total cumulative CPU time: 950 msec  
Ended Job = job\_201506301103\_0001  
Ended Job = 1716336105, job is filtered out (removed at runtime).  
Ended Job = -578743888, job is filtered out (removed at runtime).  
Moving data to: hdfs://localhost.localdomain:8020/tmp/hive-cloudera/hive\_2015-06-30\_11-22-59\_851\_2446083450544691385-1/-ext-10001  
Moving data to: hdfs://localhost.localdomain:8020/user/hive/warehouse/xml\_table  
chgrp: changing ownership of 'hdfs://localhost.localdomain:8020/user/hive/warehouse/xml\_table': User does not belong to hive  
Table default.xml\_table stats: [num\_partitions: 0, num\_files: 1, num\_rows: 0, total\_size: 46, raw\_data\_size: 0]  
2 Rows loaded to hdfs://localhost.localdomain:8020/tmp/hive-cloudera/hive\_2015-06-30\_11-22-59\_851\_2446083450544691385-1/-ext-10000  
MapReduce Jobs Launched:   
Job 0: Map: 1   Cumulative CPU: 0.95 sec   HDFS Read: 435 HDFS Write: 46 SUCCESS  
Total MapReduce CPU Time Spent: 950 msec  
OK  
Time taken: 21.649 seconds  
  
**hive> select \* from xml\_table;**                                      OK  
1 Satish Kumar Technical Lead  
2 Ramya Testing

* 1. **Hive table using Serde Regular Expression**



1. **How Table Locking Works in Hive**

This article explains how table locking works in Hive by running a series hive commands and their outputs. To do this, I have created two simple tables in my small cluster called “test” and “test\_partitioned”.

Initially, when no query is running against the “test” table, the table should have no locks:



“SHARED” lock is also called a “READ” lock, meaning, other people can still read from the table, but any writes will have to wait for it to finish.

Now if you want to write data to the table using query:

INSERT OVERWRITE TABLE test SELECT COUNT(1) FROM sample\_07;

the table will be locked “EXCLUSIVELY”

hive> SHOW LOCKS test;

OK

default@test EXCLUSIVE

Time taken: 0.158 seconds, Fetched: 1 row(s)

“EXCLUSIVE” lock is also called a “WRITE” lock, meaning no one else is able to read or write to the table while the lock is present, all other queries will have to wait for the current query to finish before they can start.

You can also manually lock tables whenever you want:

hive> LOCK TABLE test SHARED;

OK

Time taken: 0.244 seconds

hive> SHOW LOCKS test;

OK

default@test SHARED

Time taken: 0.107 seconds, Fetched: 1 row(s)

hive> UNLOCK TABLE test;

OK

Time taken: 0.255 seconds

hive> SHOW LOCKS test;

OK

Time taken: 0.114 seconds

hive> LOCK TABLE test EXCLUSIVE;

OK

Time taken: 0.154 seconds

hive> SHOW LOCKS test;

OK

default@test EXCLUSIVE

Time taken: 0.083 seconds, Fetched: 1 row(s)

hive> UNLOCK TABLE test;

OK

Time taken: 0.127 seconds

hive> SHOW LOCKS test;

OK

Time taken: 0.232 seconds

The locking can also be applied to table partitions:

hive> LOCK TABLE test\_partitioned PARTITION (p='p1') EXCLUSIVE;

OK

Time taken: 0.31 seconds

hive> SHOW LOCKS test\_partitioned PARTITION (p='p1');

OK

default@test\_partitioned@p=p1 EXCLUSIVE

Time taken: 0.189 seconds, Fetched: 1 row(s)

hive> SHOW LOCKS test\_partitioned;

OK

Time taken: 0.105 seconds

hive> UNLOCK TABLE test\_partitioned PARTITION (p='p1');

OK

Time taken: 0.136 seconds

hive> SHOW LOCKS test\_partitioned PARTITION (p='p1');

OK

Time taken: 0.123 seconds

hive> SHOW LOCKS test\_partitioned;

OK

Time taken: 0.081 seconds

When you write to a partition of a table using static partitioning, an EXCLUSIVE lock will be applied to the partition that will be written to, and SHARED lock will be applied to the table itself:

INSERT OVERWRITE TABLE test\_partitioned PARTITION (p='p1') SELECT salary FROM sample\_07;

hive> SHOW LOCKS test\_partitioned;

OK

default@test\_partitioned SHARED

Time taken: 1.345 seconds, Fetched: 1 row(s)

hive> SHOW LOCKS test\_partitioned PARTITION (p='p1');

OK

default@test\_partitioned@p=p1 EXCLUSIVE

Time taken: 0.243 seconds, Fetched: 1 row(s)

However, there is a bug in Hive that when you try to run an “INSERT OVERWRITE” using dynamic partitioning, because Hive is unable to figure out which partitions need to be locked, it currently only applies “SHARED” lock to the table being updated.

To be absolutely safe, Hive should have applied an EXCLUSIVE lock to the table to prevent any further update to the table and all partitions, but it does not.

So for the same query I ran above, I am able to run the following same query twice in two different Hive sessions and they will race with each other:

hive> INSERT OVERWRITE TABLE test\_partitioned PARTITION (p) SELECT salary, 'p1' AS p FROM sample\_07;

hive> INSERT OVERWRITE TABLE test\_partitioned PARTITION (p) SELECT salary, 'p1' AS p FROM sample\_07;

Of course, you will have to enable dynamic partitioning for the above query to run.

They can be run at the same time and whoever finishes the last will overwrite the results from the previous one.

I tested this under CDH5.3 and CDH5.4 and both of them have the bug present. There is a upstream Hive JIRA issue,  
you can have a look at [Exclusive locks are not acquired when using dynamic partitions](https://issues.apache.org/jira/browse/HIVE-3509) for more information.

1. **Importing data from MYSQL to Hive**
2. Use Sqoop to import the data from MQSQL to local linux system in the required format (.csv)
3. Create a table if not existing
4. Load the csv file into Hive table using LOAD DATA LOCAL INPATH command
5. **Incremetal Update in Hive using Sqoop**
6. Use Sqoop for Incremental update ie import incremental data from MYSQL table using Scoop

**Update support in Hive**

UPDATE tbl\_name SET upd\_column = new\_value

WHERE upd\_column = current\_value;

But to do updates in Hive you must take care of the following:

Minimum requisite to perform Hive CRUD using ACID operations is:

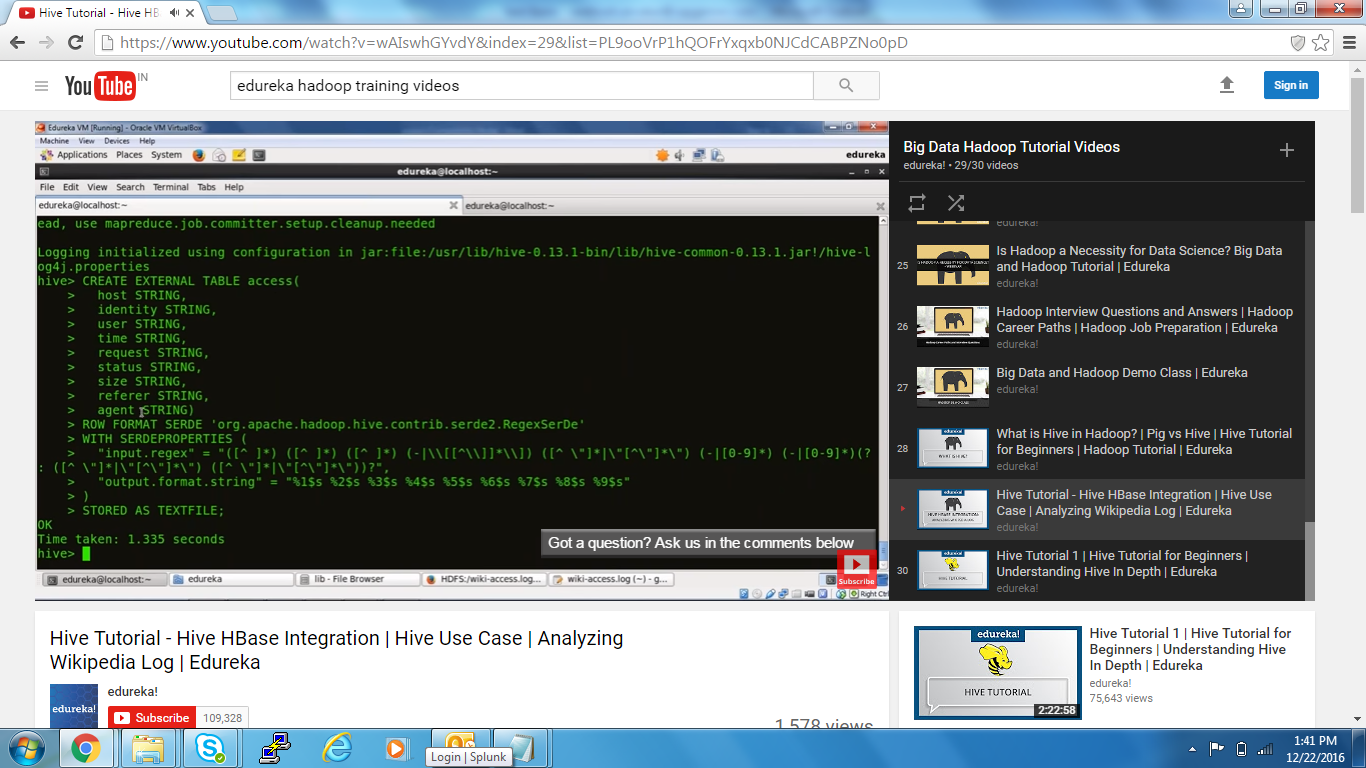
1. Hive version 0.14
2. File format must be in ORC file format with TBLPROPERTIES(‘transactional’=’true’)
3. Table must be CLUSTERED BY with some Buckets.

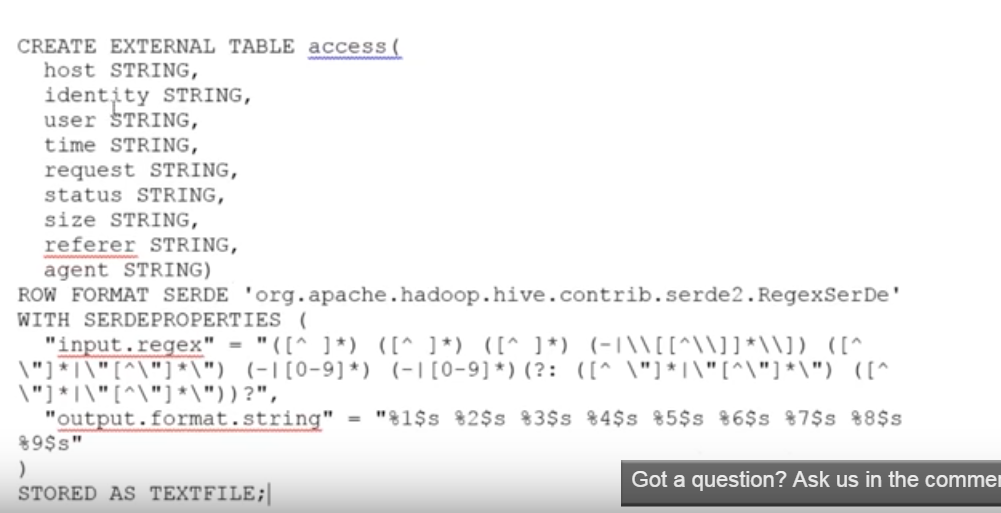
**Note:** if table is not bucketed then you will get FAILED: SemanticException [Error 10297]: Attempt to do update or delete on table student2 that does not use an AcidOutputFormat or is not bucketed.

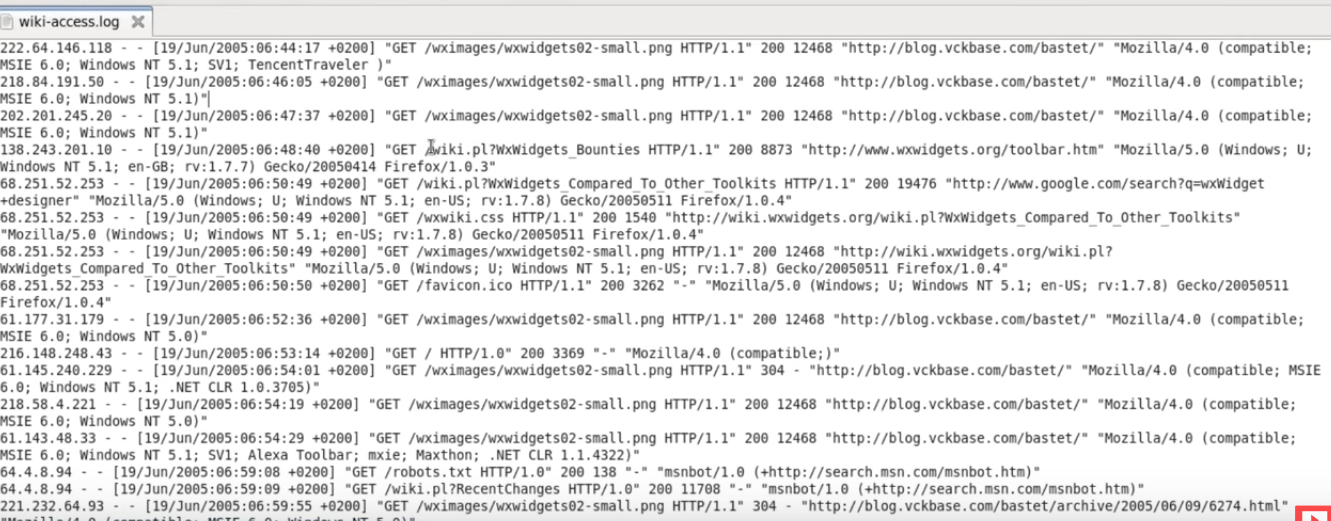
**Below are the some of new configuration parameters for supporting transactions(added in Hive 0.14):**

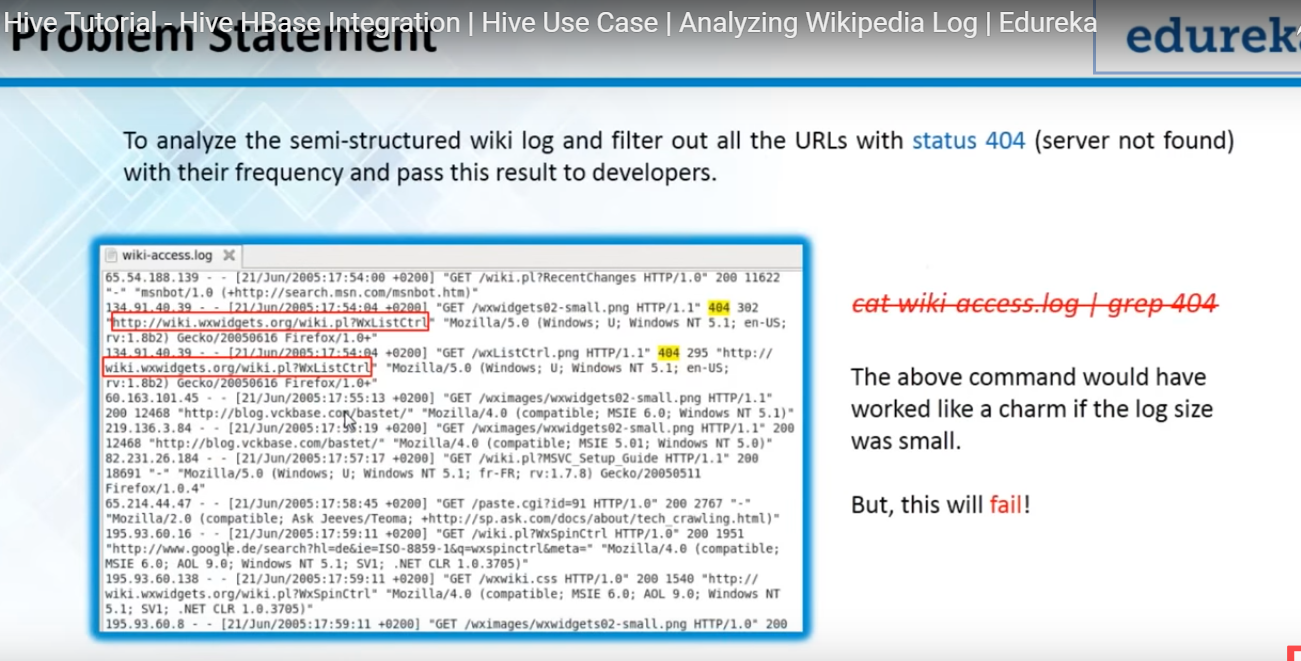
**Handling unstructured data :**

1. **Using Regular Expression Serde option**



****

****

****

**2.Using built in hive SQL functions like TRIM, SPLIT, SUBSTRING, REGEX\_REPLACE**

Below is the use case:

CREATE EXTERNAL TABLE IF NOT EXISTS access\_log (log\_line STRING)

PARTITIONED BY (hive\_entry\_timestamp STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

FIELDS TERMINATED BY '01'

STORED AS TEXTFILE

LOCATION '/user/demo/access\_logs';

As an example, here are a few lines of what a typical access log file might look like:

10.236.133.247 - - [Mon, 19 May 2014 16:31:33 GMT] "GET /api/admin/job/aggregator/status HTTP/1.1" 200 1847 "https://my.analytics.app/admin" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_9\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.137 Safari/537.36"

10.236.133.247 - - [Mon, 19 May 2014 16:31:43 GMT] "GET /api/admin/job/aggregator/status HTTP/1.1" 200 1984 "https://my.analytics.app/admin" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_9\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.137 Safari/537.36"

10.236.133.247 - - [Mon, 19 May 2014 16:33:02 GMT] "GET /dashboard/courses/1291726 HTTP/1.1" 304 - "https://my.analytics.app/admin" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_9\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.137 Safari/537.36"

10.181.145.106 - - [Mon, 19 May 2014 16:33:03 GMT] "GET /api/loggedInUser HTTP/1.1" 304 - "https://my.analytics.app/dashboard/courses/1291726" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_9\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.137 Safari/537.36"

10.181.145.106 - - [Mon, 19 May 2014 16:33:03 GMT] "POST /api/instrumentation/events/new HTTP/1.1" 200 2 "https://my.analytics.app/dashboard/courses/1291726" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_9\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.137 Safari/537.36"

ALTER TABLE access\_log

DROP IF EXISTS PARTITION (hive\_entry\_timestamp='2014-07-04T00:00Z');

ALTER TABLE access\_log

ADD PARTITION (hive\_entry\_timestamp='2014-07-04T00:00Z')

LOCATION '/user/demo/access\_logs/2014/07/04/00/00';

CREATE EXTERNAL TABLE IF NOT EXISTS parsed\_access\_log (

log\_date STRING,

ip STRING,

http\_method STRING,

uri STRING,

protocol STRING,

user\_agent STRING,

url STRING

)

PARTITIONED BY (hive\_entry\_timestamp STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '01'

STORED AS TEXTFILE

LOCATION '/user/demo/parsed\_access\_log';

ALTER TABLE parsed\_access\_log

DROP IF EXISTS PARTITION (hive\_entry\_timestamp='2014-07-04T00:00Z');

ALTER TABLE parsed\_access\_log

INSERT OVERWRITE DIRECTORY '/user/demo/parsed\_access\_log/2014/07/04/00/00'

SELECT

TRIM(SUBSTRING( log\_line, INSTR(log\_line, "[")+1, INSTR(log\_line, "]")-(INSTR(log\_line, "[")+1) )) AS log\_date,

TRIM(SUBSTRING( log\_line, 0, INSTR(log\_line, "-")-2) ) AS ip,

SPLIT(TRIM(SUBSTRING( log\_line, (INSTR(log\_line, " \"")+2), INSTR(log\_line, " \"") ) ), " ")[0] AS method,

SPLIT(TRIM(SUBSTRING( log\_line, (INSTR(log\_line, " \"")+2), INSTR(log\_line, " \"") ) ), " ")[1] AS uri,

REGEX\_REPLACE(SPLIT(TRIM(SUBSTRING( log\_line, (INSTR(log\_line, " \"")+2), INSTR(log\_line, " \"") ) ), " ")[2], "\"", "") AS proto,

TRIM(SUBSTRING(log\_line, (instr(log\_line, "\" \"")+3), ( (length(log\_line)-1)) - (INSTR(log\_line, "\" \"")+2) )) AS user\_agent,

CASE

WHEN INSTR(log\_line, "\"http") > 0

THEN TRIM(SUBSTRING( log\_line, (INSTR(log\_line, "\"http")+1), INSTR(log\_line, "\" \"") - (INSTR(log\_line, "\"http")+1) ) )

ELSE "N/A"

END AS url

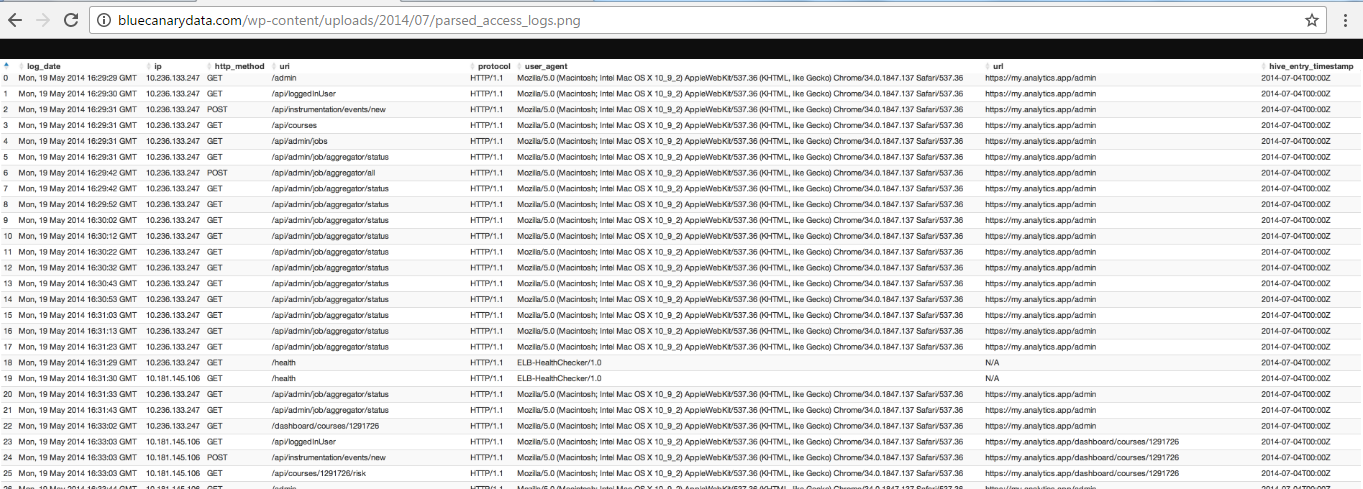
FROM access\_log

WHERE hive\_entry\_timestamp = '2014-07-04T00:00Z';

LOCATION '/user/demo/parsed\_access\_log/2014/07/04/00/00';

Finally, parse the data using Hive SQL with UDF’s:

Once the load query completes, you can select \* the processed structured version of the log data from the “parsed\_access\_log” table to get following tabular structured results:



**One of the ways identifying delta comparing current day and previous day unload file in HIVE**

**Note : Do a Full outer join of current days data with previous days data, the select fields of either todays data or previous days data based on Not null condition from the joined table where todays amount – yesterdays amount is not equal to 0.**

**How do you identify delete and inserts from the delta table ?**

**In case of deletes, the todays data will not have the row, the result of join with previous days data will have Null in todays amount column. We have a condition in select, where if amount in current days file is null, then make the amount as zero. Any row in delta file with amount as 0 is delete.**

**Rows with amount greater than zero is insert or update. The other system will query its table, if the record is found it will treat as update else as insert.**

INSERT OVERWRITE TABLE ${env}outbound.pos\_sfa\_monthly\_account\_rollup

SELECT

IF(T1.transaction\_id IS NOT NULL, T1.transaction\_id, T2.transaction\_id) as transaction\_id,

IF(T1.transaction\_type IS NOT NULL, T1.transaction\_type, T2.transaction\_type) as transaction\_type,

IF(T1.sfa\_account\_guid IS NOT NULL, T1.sfa\_account\_guid, T2.sfa\_account\_guid) as sfa\_account\_guid,

IF(T1.account\_name IS NOT NULL, T1.account\_name, T2.account\_name) as account\_name,

IF(T1.processing\_date IS NOT NULL, T1.processing\_date, FROM\_UNIXTIME(UNIX\_TIMESTAMP(),'MMddyyyy')) AS processing\_date,

IF(T1.transaction\_month IS NOT NULL, T1.transaction\_month, T2.transaction\_month) as transaction\_month,

IF(T1.product\_id IS NOT NULL, T1.product\_id, T2.product\_id) as product\_id,

IF(T1.product\_name IS NOT NULL, T1.product\_name, T2.product\_name) as product\_name,

IF(T1.gl\_currency IS NOT NULL, T1.gl\_currency, T2.gl\_currency) as gl\_currency,

CASE

WHEN T1.gl\_amount IS NOT NULL THEN T1.gl\_amount

ELSE 0

END as gl\_amount

FROM

(SELECT \* FROM ${env}sfa\_data.sfa\_rollup\_data\_stg WHERE transaction\_id IS NOT NULL AND TRIM(transaction\_id)<>'') T1

FULL OUTER JOIN

(SELECT \* FROM ${env}sfa\_data.sfa\_rollup\_data\_last\_week\_stg WHERE transaction\_id IS NOT NULL AND TRIM(transaction\_id)<>'') T2

ON (T1.transaction\_id=T2.transaction\_id)

WHERE T1.transaction\_id IS NULL OR T2.transaction\_id IS NULL OR T1.gl\_amount-T2.gl\_amount<>0;

**Dropping a column in Hive**

From this table I want to drop the column Dob. You can use the **ALTER TABLE REPLACE** statement to **drop a column.**

**ALTER TABLE test\_tbl REPLACE COLUMNS(ID STRING,NAME STRING,AGE STRING);**   you have to give the column names which you want to keep in the table

**10 Best Practices for Apache Hive**

Apache Hive is an SQL-like software used with Hadoop to give users the capability of performing SQL-like queries on it’s own language, HiveQL, quickly and efficiently . It also gives users additional query and analytical abilities not available on traditional SQL structures.

With Apache Hive, users can use HiveQL or traditional [Mapreduce](http://www.qubole.com/mapreduce-as-a-service/) systems, depending on individual needs and preferences. Hive is particularly ideal for analyzing large datasets (petabytes) and also includes a variety of storage options.

Hive is full of unique tools that allow users to quickly and efficiently perform data queries and analysis. In order to make full use of all these tools, it’s important for users to use best practices for Hive implementation. Here are 10 ways to make the most of Hive.

**1. Partitioning Tables:**

Hive partitioning is an effective method to improve the query performance on larger tables ([Tweet this](https://twitter.com/home?status=Hive%20partitioning%20is%20an%20effective%20method%20to%20improve%20the%20query%20performance%20on%20larger%20tables.%20By%20@Qubole%20http://bit.ly/1sIGoBo)). Partitioning allows you to store data in separate sub-directories under table location. It greatly helps the queries which are queried upon the partition key(s). Although the selection of partition key is always a sensitive decision, it should always be a low cardinal attribute, e.g. if your data is associated with time dimension, then date could be a good partition key. Similarly, if data has association with location, like a country or state, then it’s a good idea to have hierarchical partitions like country/state.

**2. De-normalizing data:**

Normalization is a standard process used to model your data tables with certain rules to deal with redundancy of data and anomalies. In simpler words, if you normalize your data sets, you end up creating multiple relational tables which can be joined at the run time to produce the results. Joins are expensive and difficult operations to perform and are one of the common reasons for performance issues ([Tweet this)](https://twitter.com/home?status=Joins%20are%20expensive%20and%20difficult%20operations%20to%20perform%20and%20are%20one%20of%20the%20common%20reasons%20for%20performance%20issues%20@Qubole%20http://bit.ly/1sIGo). Because of that, it’s a good idea to avoid highly normalized table structures because they require join queries to derive the desired metrics.

**3. Compress map/reduce output:**

Compression techniques significantly reduce the intermediate data volume, which internally reduces the amount of data transfers between mappers and reducers. All this generally occurs over the network. Compression can be applied on the mapper and reducer output individually. Keep in mind that gzip compressed files are not splittable. That means this should be applied with caution. A compressed file size should not be larger than a few hundred megabytes ([Tweet this](https://twitter.com/home?status=A%20compressed%20file%20size%20should%20not%20be%20larger%20than%20a%20few%20hundred%20megabytes%20By%20@Qubole%20http://bit.ly/1s)). Otherwise it can potentially lead to an imbalanced job. Other options of compression codec could be snappy, lzo, bzip, etc.

* For map output compression set **mapred.compress.map.output** to true
* For job output compression set **mapred.output.compress** to true

*For more functions, check out the*[*Hive Cheat Sheet*](http://www.qubole.com/resources/cheatsheet/hive-function-cheat-sheet/)*.*

**4. Map join:**

Map joins are really efficient if a table on the other side of a join is small enough to fit in the memory ([Tweet this](https://twitter.com/home?status=Map%20joins%20are%20really%20efficient%20if%20a%20table%20on%20the%20other%20side%20of%20a%20join%20is%20small%20enough%20to%20fit%20in%20the%20memory%20By%20@Qubole%20http://bit.ly/1s)). Hive supports a parameter, **hive.auto.convert.join**, which when it’s set to “true” suggests that Hive try to map join automatically. When using this parameter, be sure the auto convert is enabled in the Hive environment.

**5. Bucketing:**

Bucketing improves the join performance if the bucket key and join keys are common. ([Tweet this](https://twitter.com/home?status=Bucketing%20improves%20the%20join%20performance%20if%20the%20bucket%20key%20and%20join%20keys%20are%20common.%20By%20@Qubole%20http://bit.ly/1s)) Bucketing in Hive distributes the data in different buckets based on the hash results on the bucket key. It also reduces the I/O scans during the join process if the process is happening on the same keys (columns).

Additionally it’s important to ensure the bucketing flag is set (**SET hive.enforce.bucketing=true;**) every time before writing data to the bucketed table. To leverage the bucketing in the join operation we should **SET hive.optimize.bucketmapjoin=true**. This setting hints to Hive to do bucket level join during the map stage join. It also reduces the scan cycles to find a particular key because bucketing ensures that the key is present in a certain bucket.

**6. Input Format Selection:**

Input formats play a critical role in Hive performance. For example JSON, the text type of input formats, is not a good choice for a large production system where data volume is really high. ([Tweet This](https://twitter.com/home?status=JSON%20is%20not%20a%20good%20choice%20for%20a%20large%20production%20system%20where%20data%20volume%20is%20really%20high%20By%20@Qubole%20http://bit.ly/1s)) These type of readable formats actually take a lot of space and have some overhead of parsing ( e.g JSON parsing ). To address these problems, Hive comes with columnar input formats like RCFile, ORC etc. Columnar formats allow you to reduce the read operations in analytics queries by allowing each column to be accessed individually. There are some other binary formats like [Avro](http://qubole-documentation.readthedocs.org/en/latest/faqs/hive/file-formats-quboles-hive-support-box.html?highlight=avro), sequence files, Thrift and ProtoBuf, which can be helpful in various use cases too.

**7. Parallel execution:**

Hadoop can execute MapReduce jobs in parallel, and several queries executed on Hive automatically use this parallelism. However, single, complex [Hive queries](https://www.qubole.com/resources/webinars/build-your-customer-hive-udfs/) commonly are translated to a number of MapReduce jobs that are executed by default sequencing. Often though, some of a query’s MapReduce stages are not interdependent and could be executed in parallel. They then can take advantage of spare capacity on a cluster and improve cluster utilization while at the same time reducing the overall query executions time. The configuration in Hive to change this behavior is merely switching a single flag **SET hive.exec.parallel=true**.

**8. Vectorization:**

Vectorization allows Hive to process a batch of rows together instead of processing one row at a time. ([Tweet This](https://twitter.com/home?status=Vectorization%20allows%20Hive%20to%20process%20a%20batch%20of%20rows%20together%20instead%20of%20processing%20one%20row%20at%20a%20time.%20By%20@Qubole%20http://bit.ly/1s)) Each batch consists of a column vector which is usually an array of primitive types. Operations are performed on the entire column vector, which improves the instruction pipelines and cache usage. To enable vectorization, set this configuration parameter **SET hive.vectorized.execution.enabled=true**.

**9. Unit Testing:**

Simply speaking, unit testing determines whether the smallest testable piece of your code works exactly as you expect. Unit testing gives a couple of benefits i.e. detecting problems early, making it easier to change and refactor code, being a form of documentation that explains how code works, to name a few.

In Hive, you can unit test UDFs, SerDes, streaming scripts, Hive queries and more. To a large extent, it is possible to verify the correctness of your whole HiveQL query by running quick local unit tests without even touching a Hadoop cluster. Because executing HiveQL query in the local mode takes literally seconds, compared to minutes, hours or days if it runs in the Hadoop mode, it certainly saves huge amounts of development time.

There are several tools available that helps you to test Hive queries. Some of them that you might want to look at [HiveRunner](https://github.com/klarna/HiveRunner" \t "_blank), [Hive\_test](https://github.com/edwardcapriolo/hive_test" \t "_blank)and [Beetest](https://github.com/kawaa/Beetest" \t "_blank).

**10. Sampling:**

Sampling allows users to take a subset of dataset and analyze it, without having to analyze the entire data set. If a representative sample is used, then a query can return meaningful results as well as finish quicker and consume fewer compute resources.

Hive offers a built-in TABLESAMPLE clause that allows you to sample your tables. TABLESAMPLE can sample at various granularity levels – it can return only subsets of buckets (bucket sampling), or HDFS blocks (block sampling), or only first N records from each input split. Alternatively, you can implement your own UDF that filters out records according to your sampling algorithm.

**11.TECHNIQUE #1: USE TEZ**

Hive can use the [Apache Tez](http://hortonworks.com/hadoop/tez) execution engine instead of the venerable Map-reduce engine. I won’t go into details about the many benefits of using Tez which are mentioned here; instead, I want to make a simple recommendation: if it’s not turned on by default in your environment, use Tez by setting to ‘true’ the following in the beginning of your Hive query:

Set hive.execution.engine=tez;  
With the above setting, every HIVE query you execute will take advantage of Tez.